



Team Controller
Northern Arizona University
Flagstaff, Arizona
April 7th, 2024

Zachary Parham (Team Lead): zip29@nau.edu
Tayyaba Shaheen (Mentor): ts2434@nau.edu
Bradley Essegian: bbe24@nau.edu
Brandon Udall: bcu8@nau.edu
Dylan Motz: djm658@nau.edu

Developer Guide

Northrop Grumman

Weapon System Support Software

Table of Contents

| | |
|--|----------|
| 1.0 Setting Up Your Environment..... | 3 |
| 1.1 Obtaining the Source Code..... | 3 |
| 1.2 Obtaining QT..... | 3 |
| 2.0 Understanding the Architecture..... | 4 |
| 3.0 QT Designer Basics..... | 4 |
| 4.0 Testing In QT..... | 4 |
| 4.1 Project Organization..... | 4 |
| 4.2 Test Architecture..... | 5 |
| 4.3 Running Tests..... | 5 |
| 5.0 Packaging For Release..... | 6 |

1.0 Setting Up Your Environment

Following the steps in this section will ensure you have everything you need to begin development on this project.

1.1 Obtaining the Source Code

Option 1 Github Fork:

The easiest way to obtain the source code is by forking our repository so you have a complete copy of the project (including all released versions, tests, issues, milestones and documentation) within your own new repository. After forking, be sure to change the visibility to private. Once complete you will become the owner of the new forked repository and will have full access to clone, modify and share however you wish. Team Controller will have no access or authority over the forked repository.

Option 2 Direct Download:

Of course you could also just download the files you need directly from our repository and avoid using GitHub altogether. The folder containing all necessary source code (excluding tests) is [here](#).

1.2 Obtaining QT

QT Background Information:

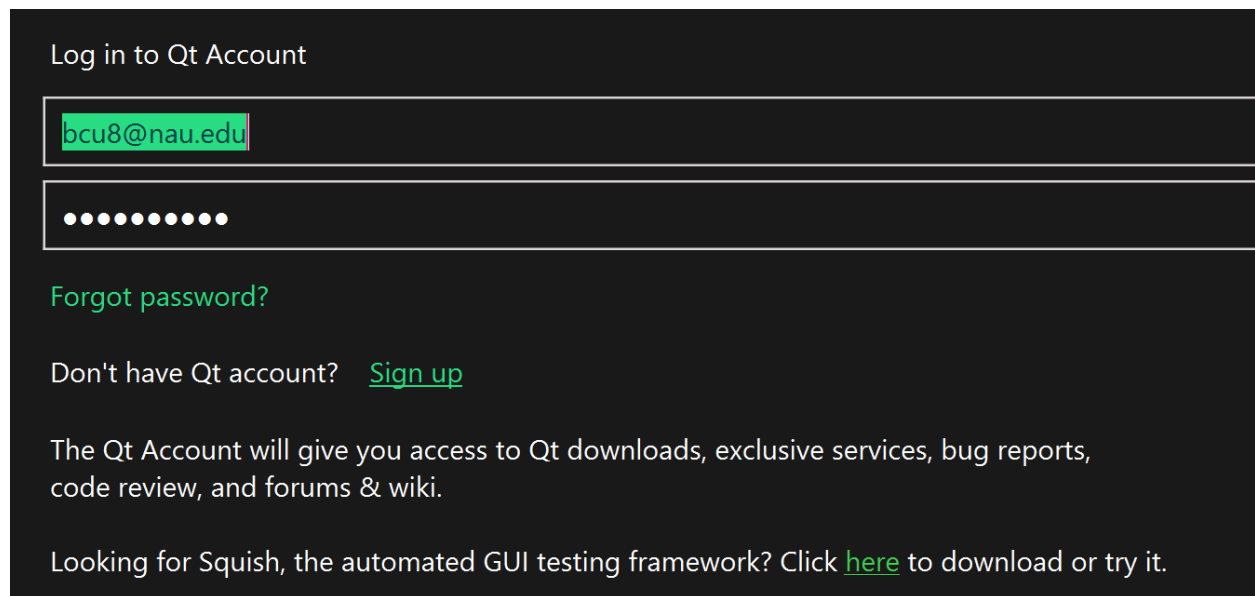
QT comes in 2 editions, there is a community edition which is free and a commercial edition which is paid. There is **one restriction** with the QT community edition. It requires all QT libraries to be dynamically linked if your code is proprietary (which in this case it is).

If you want to statically link the libraries you must use the LGPL license on your code which makes it open source. The difference being that statically linked libraries are compiled directly into your executable file while dynamically linked libraries are contained in separate files with a dll (dynamically linked library) file extension and referenced at runtime by your executable.

In our current implementation, the libraries are dynamically linked which allows you to use whatever license you want without any further restrictions or requirements.

Installing QT Community Edition:

To install the community edition visit this link.. [QT Community Edition](#). After downloading and launching the installer you will be asked to login or create a Qt account.

A screenshot of the Qt Account login interface. The background is dark gray. At the top, the text "Log in to Qt Account" is displayed in white. Below this, there are two input fields. The first field contains the email address "bcu8@nau.edu" in white text. The second field contains a series of ten white dots, representing a password. Below the password field, the text "Forgot password?" is shown in a light blue color. Further down, the text "Don't have Qt account?" is followed by a light blue link "Sign up". At the bottom, a paragraph of white text states: "The Qt Account will give you access to Qt downloads, exclusive services, bug reports, code review, and forums & wiki." Below this, another line of white text says: "Looking for Squish, the automated GUI testing framework? Click [here](#) to download or try it."

Next you will be asked to review the licensing agreement. Feel free to take the time to do your own research on the licensing restrictions involved with Qt community edition but the rundown is given in the previous section.

Qt Open Source version is available under GNU General Public License v3 and Lesser GNU General Public License v3. A few components are available under GNU General Public License v2.
Read and accept the Open Source Usage Obligations below. Reading the link below helps you choosing the right license for your project.

[Choosing the right license for your projects](#)
[Buy Qt](#)

(Lesser) GNU General Public License v3 obligations

- You must not combine code developed with a commercial Qt license with code developed with an open source license of Qt in one project or product
- Provide a license copy & explicitly acknowledge Qt use
- Make a Qt source code copy available for customers
- Accept that Qt source code modifications are non-proprietary
- Make consumer devices, which allow users to access to install and run modified versions of the SW inside them
- Accept Digital Rights Management terms, see the [GPL FAQ](#)
- Take special consideration when attempting to enforce software patents [FAQ](#)
- For further information, refer to [GPLv3](#) and [LGPLv3](#)

☒ I have read and agree to the terms and conditions of using Open Source Qt

Team Controller

☐ I'm an individual and do not use Qt for any company

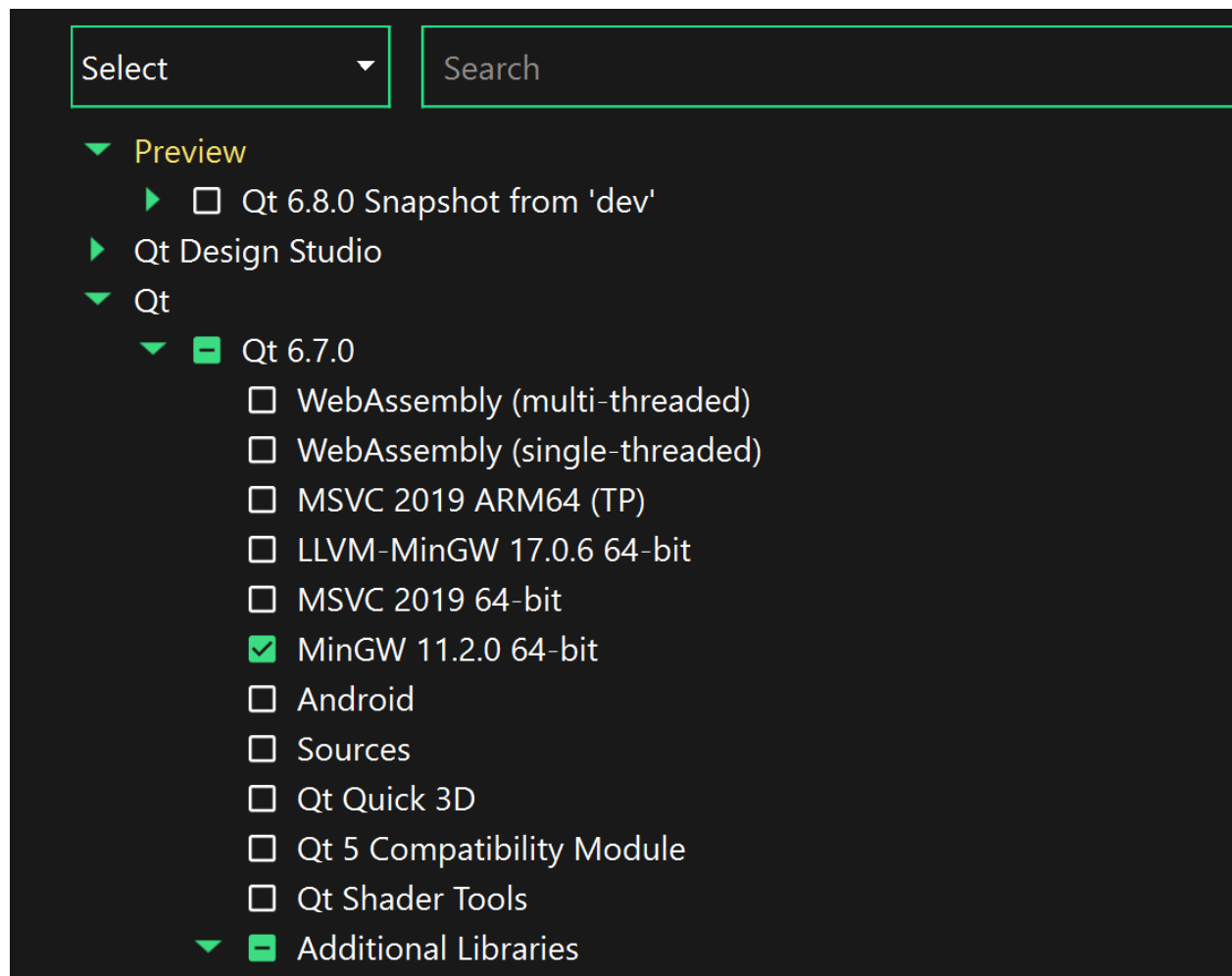
< Back

Next >

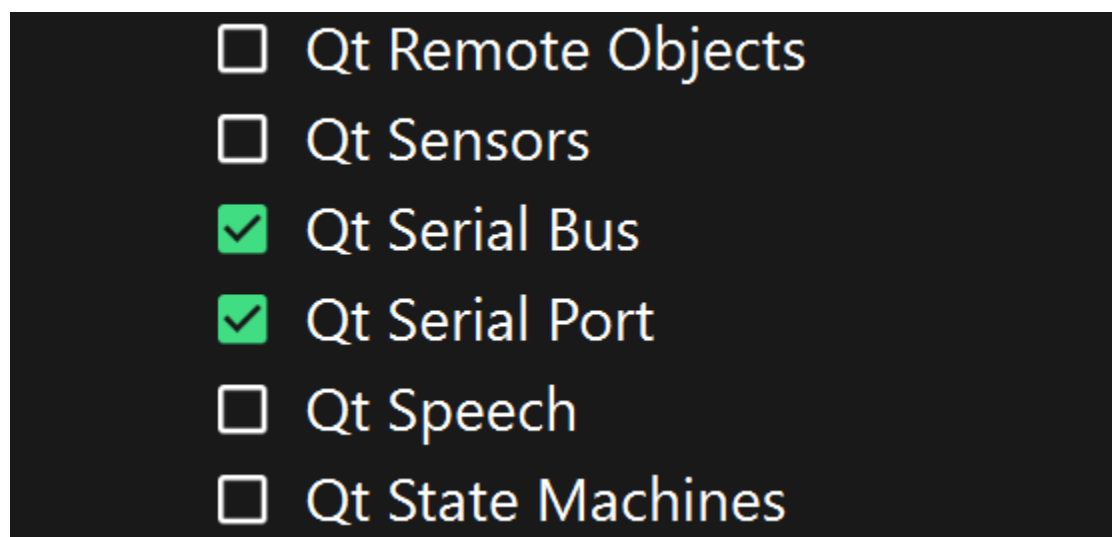
Cancel

Click next twice then choose your desired installation folder for Qt and all of its components.

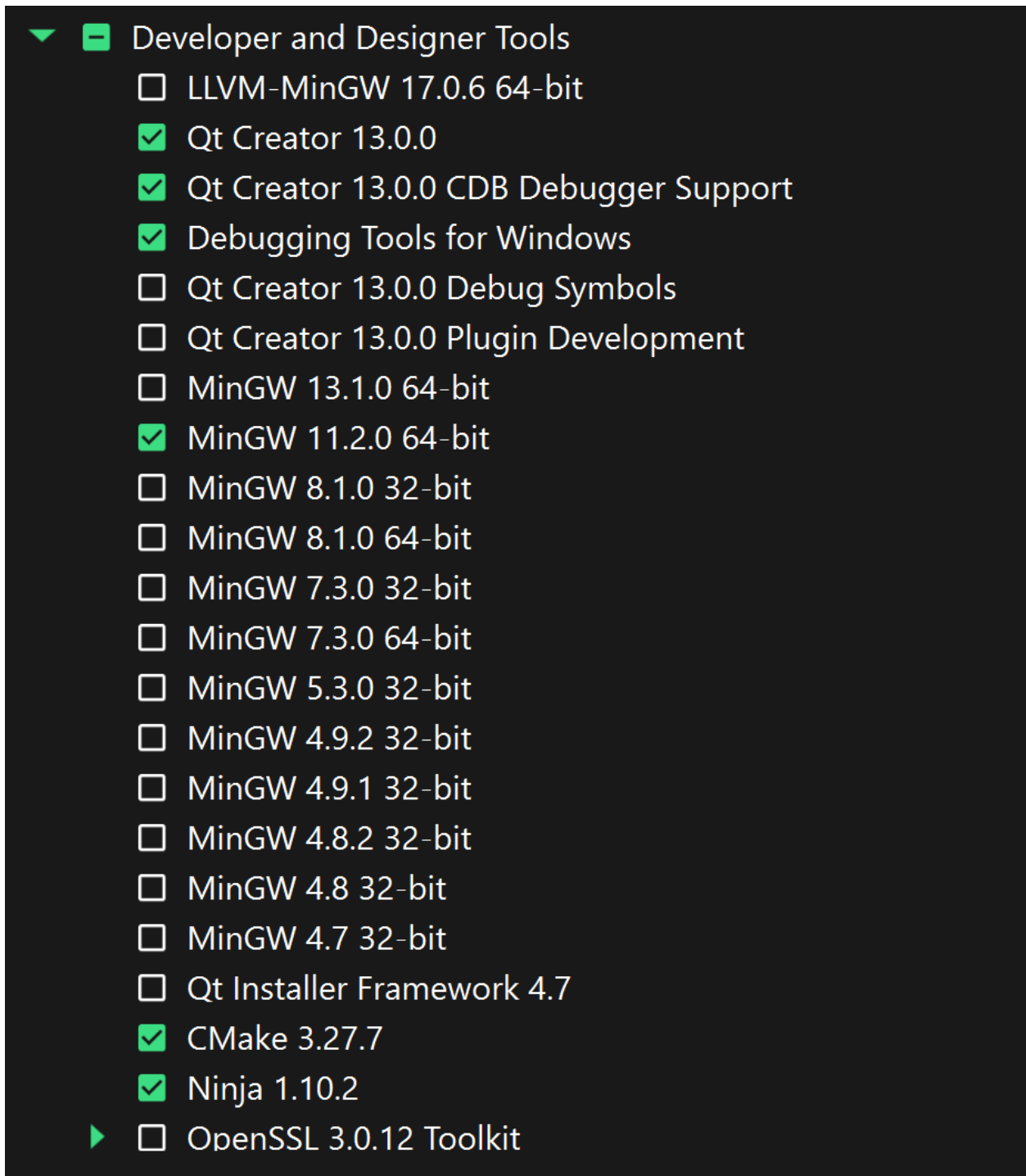
Once you get to the **select components** page, it is essential that you select QtSerialPort, and QtSerialBus libraries as well as the compiler you would like to use. Expand Qt, then expand the latest version, select your desired compiler (we used and recommend MinGW) then expand additional libraries.



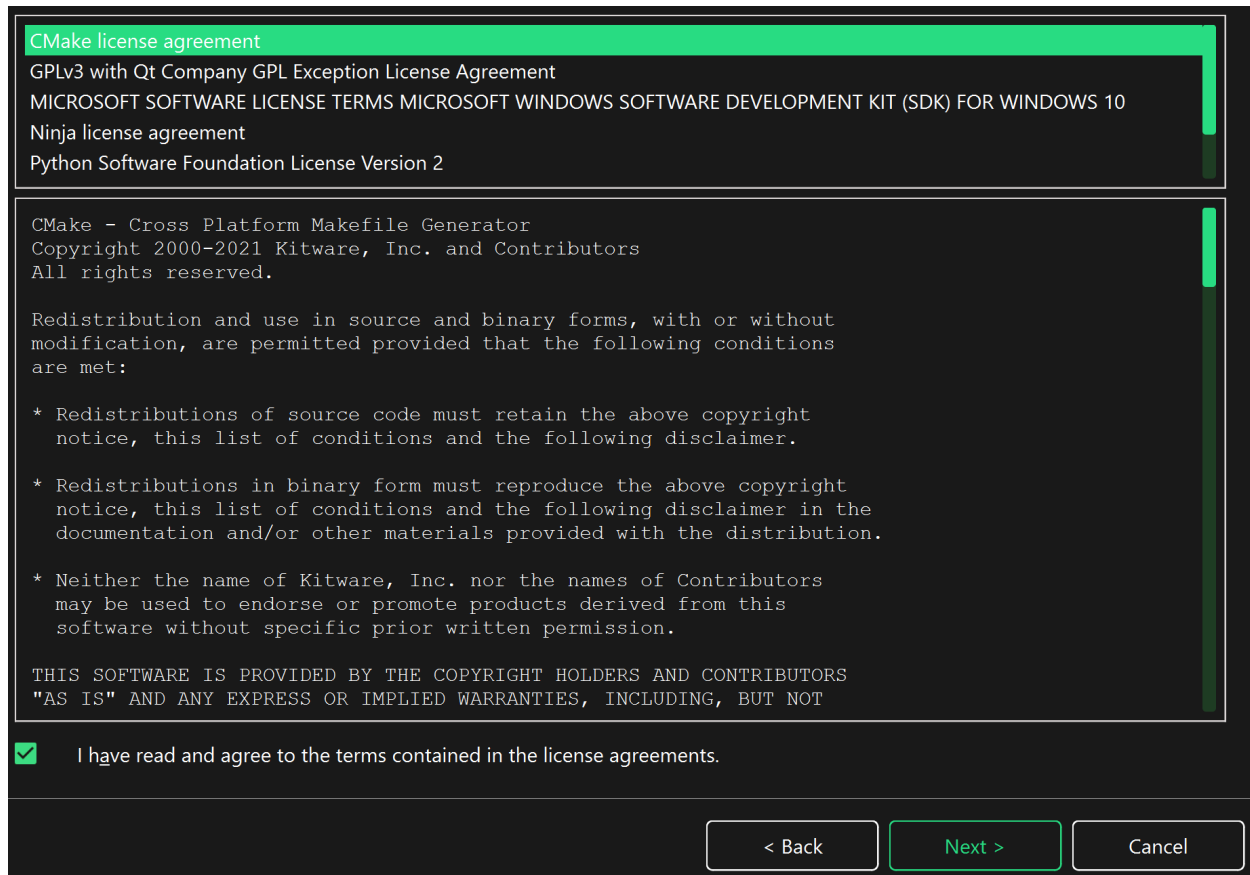
Scroll down until you find the components.



Under developer and designer tools, select the same version of compiler you selected previously and leave the rest of the options as shown.



Agree to the CMake license agreement, again do further research as needed.

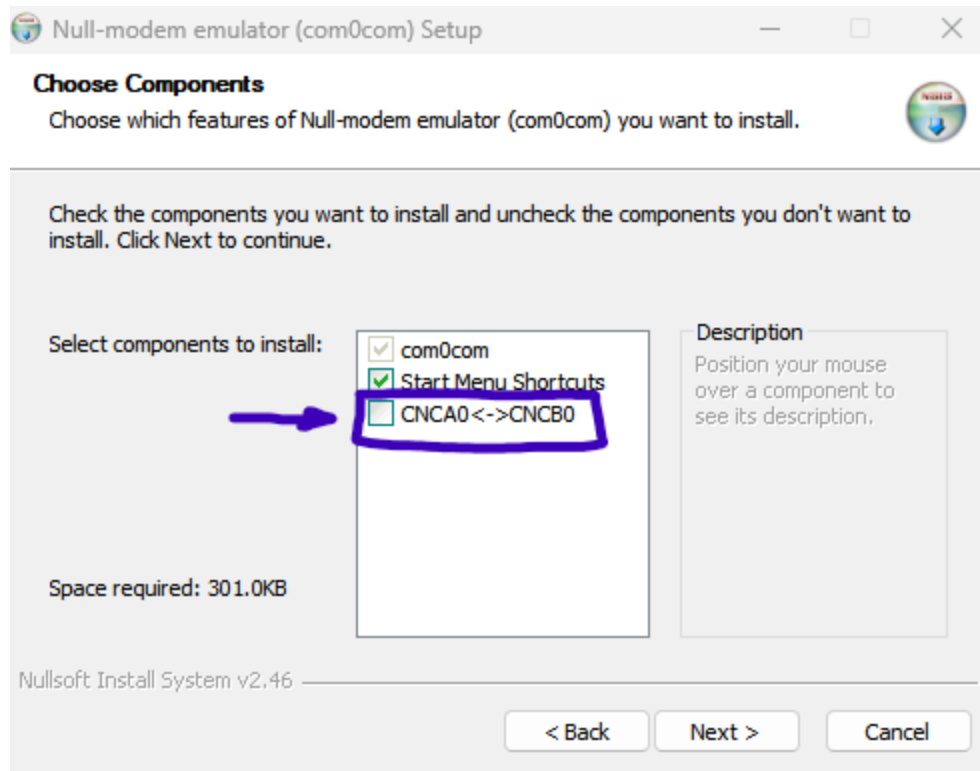


Click next twice to begin the installation. Once complete you can manage your installation, including downloading Qt software updates, installing additional libraries, and uninstalling Qt from the **Qt Maintenance Tool** inside the Qt installation folder.

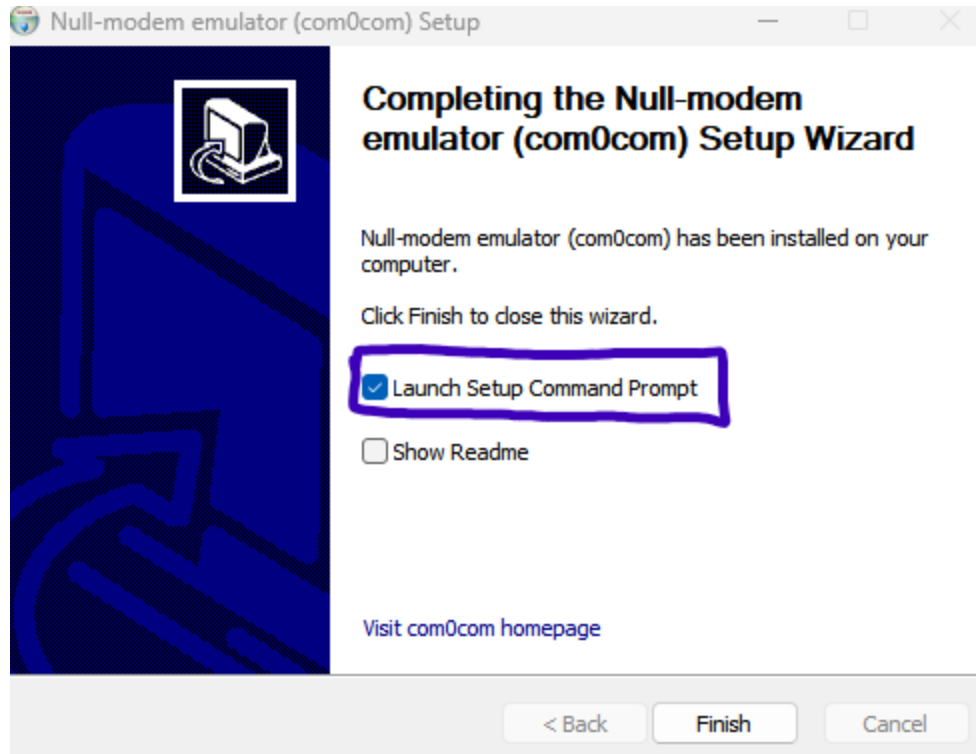
1.3 [OPTIONAL] Installing Virtualized Serial Ports

For testing serial communication, we often used virtualized serial ports to facilitate communication between our controller simulator and our application without the need for serial hardware. To do this you must install 2 serial drivers and a serial bus to connect them. We used an application called [com0com](#).

Once you launch the installer, we recommend you uncheck the box labeled **CNCA0<->CNCB0**. It is meant to automatically set up the ports but we have experienced issues using this feature.



Continue with the setup until you get to the “**Finish**” page. Here you will check “**Launch Setup Command Prompt**” then finish. In the command prompt you can type “help” to get a list of commands to set up and customize your ports.



For example in the command prompt you can enter the following commands (with your desired port names) to create the virtualized ports and check if the process was successful.

```
command> install PortName=COM9 PortName=COM10
          CNCA2 PortName=COM9
          CNCB2 PortName=COM10
ComDB: COM9 - logged as "in use"
ComDB: COM10 - logged as "in use"
command> list
          CNCA0 PortName=-
          CNCB0 PortName=-
          CNCA1 PortName=COM4
          CNCB1 PortName=COM5
          CNCA2 PortName=COM9
          CNCB2 PortName=COM10
```

After this process, your virtualized ports will be accessible by our application and you can close the command prompt. If you experience issues, you can visit online forums and/or the ReadMe documentation for troubleshooting help.

To use these ports it is important to understand the following about our architecture. The CSim class is designed to launch a thread then access the serial port selected in the dev page. Once you click start CSim, it will open its serial port and begin listening for a handshake message, meanwhile generating data that will be dumped once a connection occurs. To create the

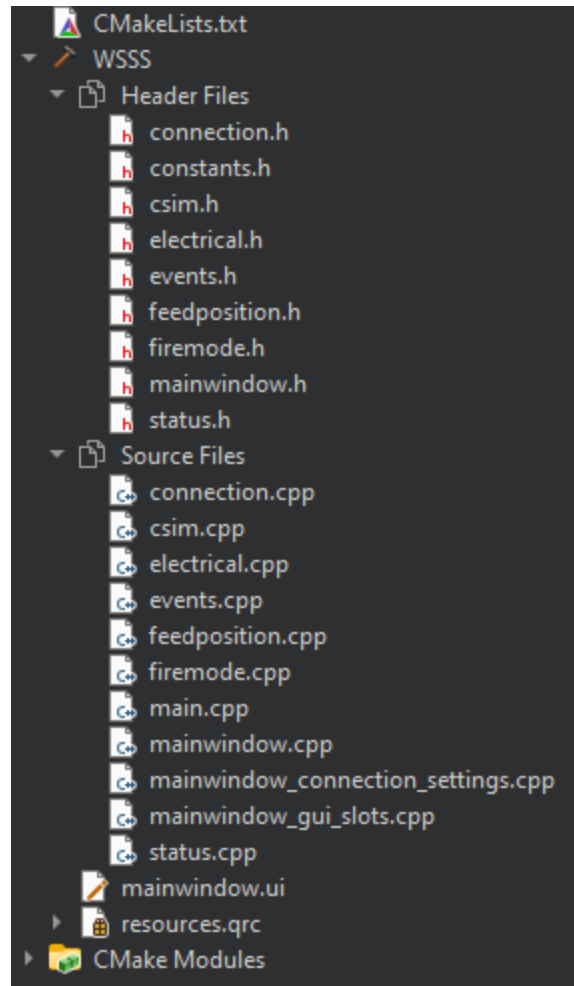
connection, you must go to the connection page and select the counterpart of CSim's virtual serial port then click connect. Once the handshake occurs data will be continuously read from CSim to the application through the virtual serial ports you created until you disconnect from the connection page or stop CSim from the developer page.

2.0 Understanding the Application Architecture

2.1 Introduction

We designed our system to be modular and object oriented using C++ features enhanced by the QT framework and libraries. Each class has a header file which defines its variables and methods, and a cpp file which contains the implementation. With the exception of constants.h (which defines constants used in various places) and the MainWindow class which has its implementation split up into 3 cpp files containing the implementation of different aspects of the class. For a closer look at the interaction between classes and the serial communication protocol we used, take a look at our class and sequence diagrams respectively. These can be found in the [diagrams](#) directory of our github. Note that these diagrams are simplified to give a general understanding without overcrowding with information, thus a complete understanding can only come from reading the source code and its documentation.

Below is the source code file structure of our project. In the header file of each class you will find a general overview of how that class ties into the overall program and the essential operations it serves.



MainWindow is the heart of the program, it manages the GUI as well as the underlying processing by declaring the various classes and managing them based on input from the controller and the user. During our development we created a threaded class called CSim which generates data to send to MainWindow. CSim is also managed by MainWindow which allows the user (in this case a developer) to send test data as if it was the controller. Since our team was not given access to the serial protocol used by Northrop Grumman, unfortunately changes will need to be made to the serial protocol for this application to work with real weapon controllers.

The next section will attempt to assist the engineers responsible for making these changes by introducing some of the portions of the code that will need to be modified.

We are assuming that Northrop Grumman has its own method of getting serial data to MainWindow, thus we will not cover anything related to CSim from this point forward. If you want to remove the CSim portions of the code please see section 2.4 Compilation Options for instructions on how to do this.

2.2 Necessary Modifications

Based on our estimates, below are the functions that need to be modified before application use. **Red indicates that the function WILL need to be modified.** Black indicates that the function MAY need to be modified. It is recommended that you reference the source code directly for detailed information on each class and function as it is heavily documented and uses clean coding conventions.

Needs Modification:

- Mainwindow
 - readSerialData() *contained within mainwindow.cpp*
 - handshake() *contained within mainwindow.cpp*
 - on_handshake_button_clicked() *contained within mainwindow_gui_slots.cpp*
- Status
 - loadData()
 - loadVersionData()
- Firemode
 - paintEvent()
- Feedposition
 - paintEvent()
 - event()
- Events
 - loadEventData()
 - loadErrorData()
 - loadEventDump()
 - loadErrorDump()
 - nodeToString()
 - stringToNode()
 - addError()
 - addEvent()
- Electrical
 - loadElecData()
 - loadElecDump()
- Connection
 - sendDisconnectMsg()
 - sendHandshakeMsg()
 - checkForValidMessage()
 - transmit()
- Constants.h

Most of these **modifications revolve around the serial communication aspects of our program**. Though since we were not given exact values to use for variables such as feedPosition firingMode and processorState, **front end changes may have to be made as well**. Assuming we have the correct data and functionality contained in each class, the rest of our program should work as intended once these load data functions have been adjusted to work with the proper serial communication protocol and the proper changes are made to the GUI to display relevant values for feed position, processor state, and firing mode (more on this later).

Our protocol is currently reminiscent of UDP, there is no flow control and it is impossible in most cases to tell if the data was transmitted correctly. We chose not to invest into a strong protocol as we knew it would be overwritten once our product was passed off. We rather focused on fleshing out some of the other aspects of the program such as the backend methods and GUI features.

In the case that changes need to be made to the text on the status page, you can use the constants.h file to change the enumerated values for variables such as ControllerState, FiringMode, FeedPosition, etc. For enumerated values such as these which are relevant to the GUI display, a constant array of strings is provided which alters the text displayed for each of the enumerated types. If you make changes to the enumeration of a certain type, be sure to also change the NUM constant associated with that type.

Making these adjustments (along with some minor adjustments to the areas in which these enumerated values are used) will automatically cause the necessary changes in the GUI, however additional work may be required to make the layout look good while displaying the changed values. For an introduction to the mainwindow.ui file and the QT GUI Designer, please refer to section 3.

It was a major focus of ours to make this program easily adaptable to such changes. Our constants.h file provides an easy method of changing key aspects. Though **sometimes more than one area of code must be altered** to complete the change (especially with GUI changes).

2.3 QT Libraries and Important Features

As mentioned previously, our application is heavily reliant on the Qt environment. Some of the integral libraries/features include.

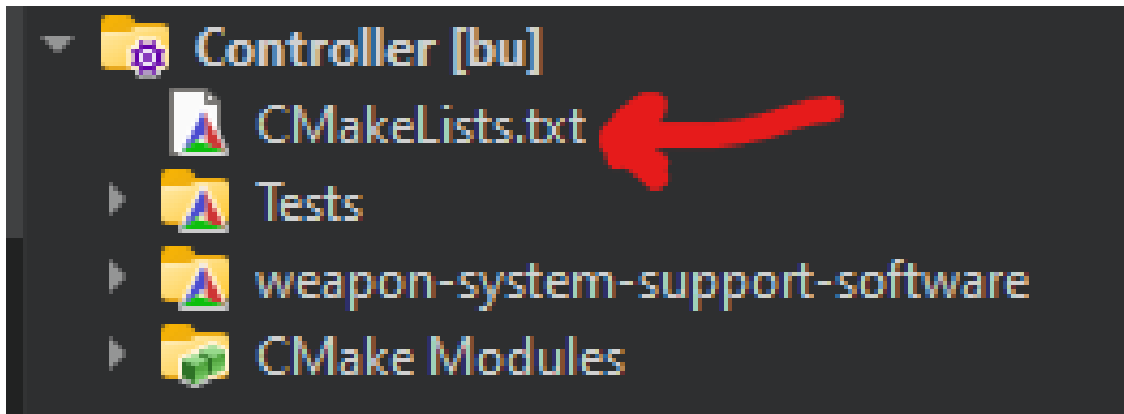
- **QtSerialPort** (provides methods for managing and interfacing with serial ports on the client computer, usage of this class can be seen most clearly in the Connection class as well as the mainwindow_connection_settings.cpp file)

- **QString** (adds high level functionality for managing string objects, this is used essentially everywhere)
- **QSettings** (for storing settings for cross session use in the windows/linux registry. Used by mainwindow to store connection settings from the connection page as well as application settings from the settings page. Search for “userSettings” to find everywhere this library is utilized)
- **QObject** (for managing the GUI elements from the back end. This is used to alter the state of the GUI widgets and is most prominently found in the mainwindow_gui_slots.cpp file)
- **Signals and Slots** (a framework which involves connecting signals to certain functions called slots. Essentially when a signal is emitted, the connected slot function is run. This is used by our readSerialData slot which is connected to the readyRead() signal from QtSerialPort. A signal is also used for all GUI inputs; you can find the slots for these signals in the mainwindow_gui_slots.cpp file. Finally our timers rely on slots to perform actions based on time such as handshake and timeout. Signals are also an easy way of calling functions in the parent class, for example a signal in the Events class can be connected to a slot in the MainWindow class which will essentially allow the Events class to trigger functions declared outside its scope.)
- **QTest** (Used for unit testing; found within the Tests directory)
- **StyleSheets** (Used to define the visual effects for QWidgets which are essentially GUI elements. These can define style changes for events such as hover (when cursor hovers over the widget) or disabled.)

QT has excellent documentation on each of its classes. The documentation home page can be found at <https://doc.qt.io/qt.html>. You can use the search feature to find any class or use the navigation buttons.

2.4 Compilation Options

In the root CMakeLists file we have a compiler directive called DEV_MODE. When its value is set to 0 all CSim related methods and variables will be prevented from compiling.

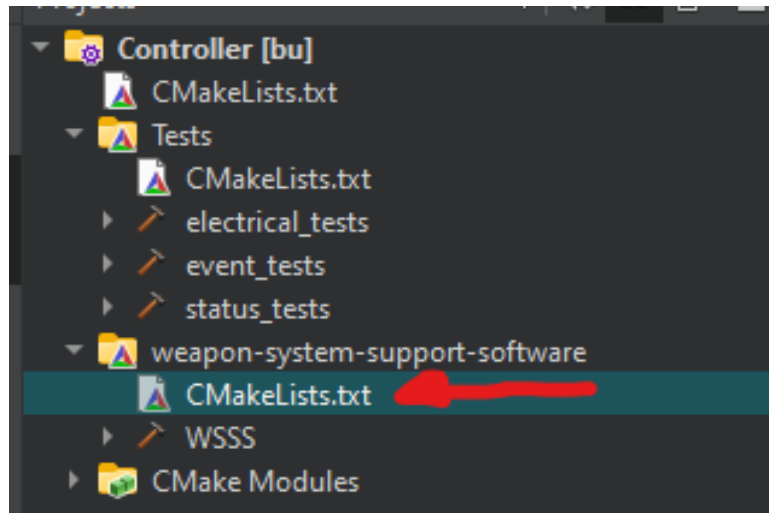


```
#set DEV_MODE to 1 to compile for dev release  
#set DEV_MODE to 0 to compile for user release  
add_compile_definitions(DEV_MODE=1)
```

If you want to remove the CSim components from the source code entirely, simply remove all source code contained within the following pre-compiler directives.

```
#if DEV_MODE  
...  
#endif
```

In the source code folder, there are additional compilation options set for conditionally compiling various `QDebug()` outputs which are printed during the program's operation in the QtEnvironment. We have also seen that `QDebug()` behaves identically to `cout` when the program is run from the command line (aka outside the Qt environment).



```
#set to 1 to enable qdebug outputs for specified components
#some messages (such as errors) deemed essential are exempt from conditional compiling
add_compile_definitions(SERIAL_COMM_DEBUG=0)
add_compile_definitions(CSIM_DEBUG=0)
add_compile_definitions(EVENTS_DEBUG=0)
add_compile_definitions(STATUS_DEBUG=0)
add_compile_definitions(GUI_DEBUG=1)
add_compile_definitions(GENERAL_DEBUG=1)
```

Below is an example of a non-error qDebug() output being set to conditionally compile based on dev mode and serial comm. We were unsure of the effect these output statements would have on performance if left in the user release which is why we made them only compile in DEV_MODE.

```
#if DEV_MODE && SERIAL_COMM_DEBUG
QDebug() << "Message sent through " << portName << " : " << message << Printable("\n");
QDebug() << bytesWritten << " bytes written to the serial port.";
#endif
```

The error outputs are left as is, without conditional compilation.

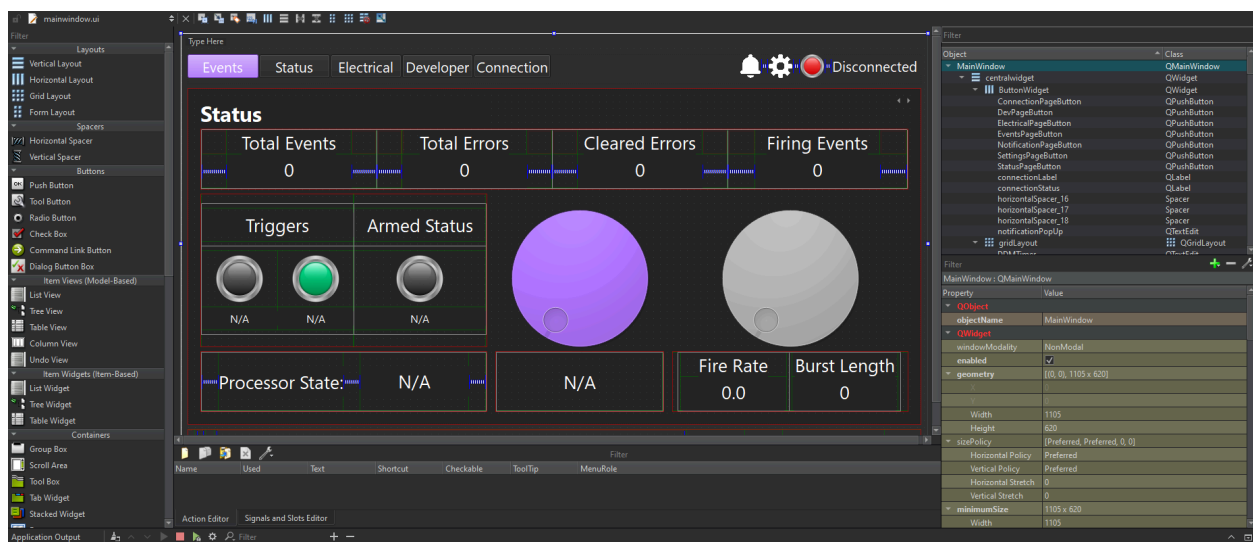
```
if (!serialPort.isOpen())
{
    // notify
    qDebug() << "Failed to open " << portName;
}
```

Due to the sheer number of `qDebug()` calls we have in our program the terminal output in the Qt environment can get cluttered with all of these active, especially the serial communication option. So this structure helps reduce clutter which makes it easier to debug certain portions of the code. While also unconditionally listing any errors which were detected throughout the execution.

3.0 QT Designer Basics

Designing a GUI is possible with the QT designer. Here is a link to the full designer manual: <https://doc.qt.io/qt-6/qtdesigner-manual.html>.

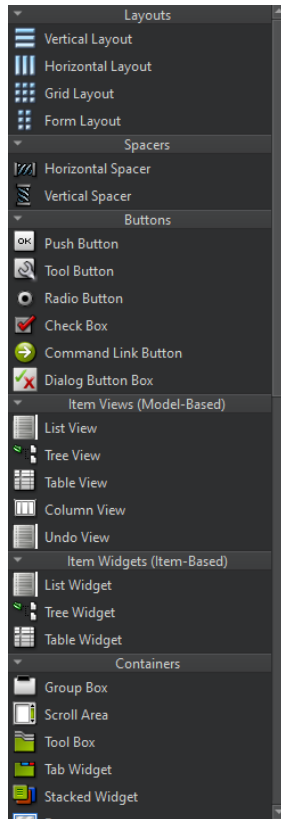
Below is an image of the QT designer. To access it, click on the `mainwindow.ui` file from the pane on the left of QT's IDE.



The `mainwindow.ui` file contains managed XML code which means you can't alter the code directly. Alterations must come from changes made using the Designer interface.

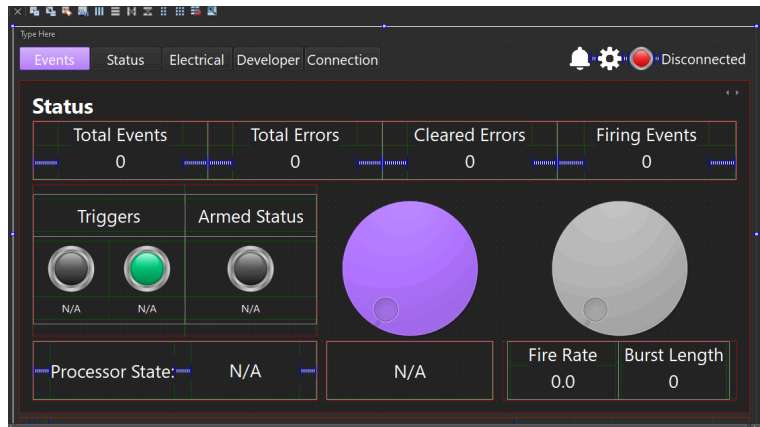
4.1 Designer Objects

On the left side of the screen is a list of all the objects that can be dragged and dropped onto the window. One of the most important objects to mention are layouts which help organize other objects and help with resizing.



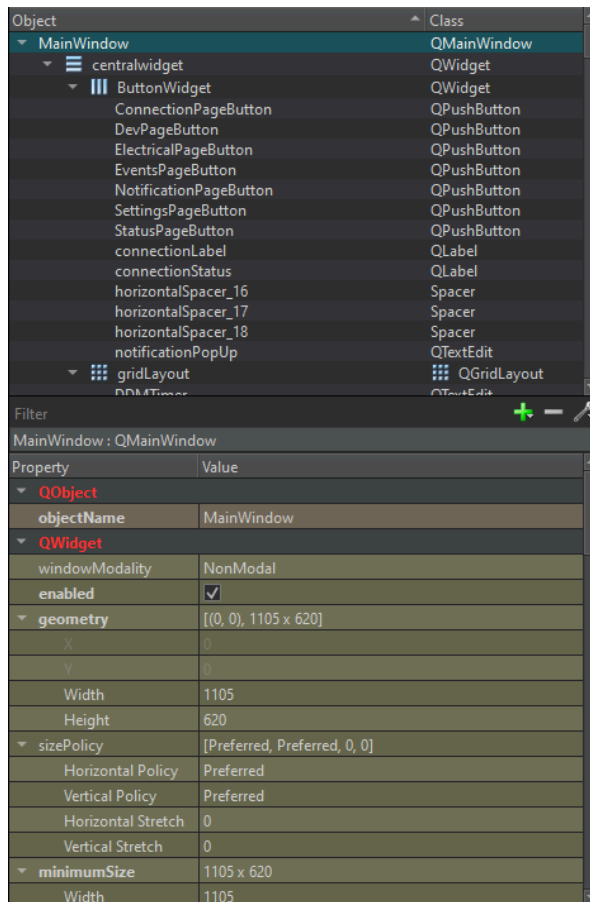
4.2 Main Window

In the middle of the screen is a window which you're able to customize how the program will look. It is also very simple and you are able to drag and drop any object from the left bar onto the main window.



4.3 Objects and properties

On the right side of the screen you will see objects and properties. This is where you see a list of all the objects you have placed on the main window and it shows how the objects are organized. You are also able to select a specific object and edit its properties and these properties include sizing and more.



4.4 Style Sheets

Style sheets are just like CSS where you can further customize an object. This can be accessed in either the properties area or by right clicking an object and selecting change stylesheet. There is a long list of what stylesheets can do including changing color, font, padding and more. A more detailed list can be found here: <https://doc.qt.io/qt-6/stylesheet-reference.html>.

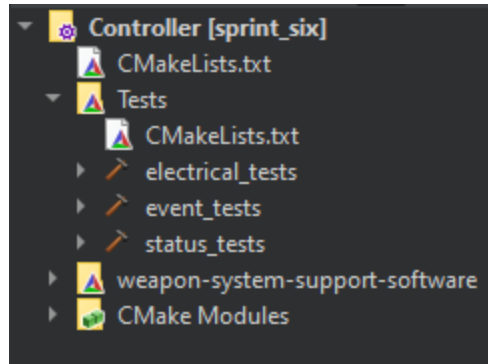
| StatusPageButton : QPushButton | |
|--------------------------------|--|
| Property | Value |
| mouseTracking | <input checked="" type="checkbox"/> |
| tabletTracking | <input type="checkbox"/> |
| focusPolicy | StrongFocus |
| contextMenuPolicy | DefaultContextMenu |
| acceptDrops | <input type="checkbox"/> |
| ▶ tooltip | |
| tooltipDuration | -1 |
| ▶ statusTip | |
| ▶ whatsThis | |
| ▶ accessibleName | |
| ▶ accessibleDescription | |
| layoutDirection | LeftToRight |
| autoFillBackground | <input type="checkbox"/> |
| stylesheet | QPushButton {\n color: rgb(255, 255, 25... |
| ▶ locale | English, United States |
| ▶ inputMethodHints | ImhNone |
| ▼ QAbstractButton | |

4.0 Testing In QT

Testing within the application is possible with the Qt Test library. A link to the documentation can be found here: <https://doc.qt.io/qt-6/qtest-overview.html>.

4.1 Project Organization

The project is organized in the following structure:



Screenshot depicting the project organization

The Controller project, with included CMakeLists.txt file, runs the entire application. Nested beneath the main CMakeList file is the Test and main application projects. Inside the Tests project are individual executables to run each unit test script.

4.2 Test Architecture

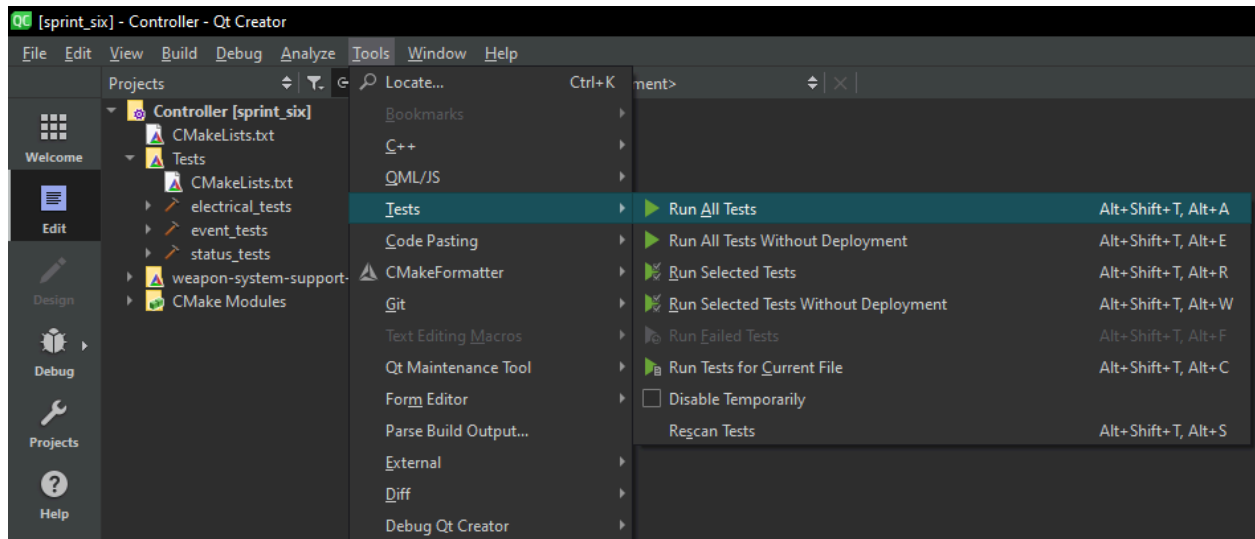
To write tests we are using two main functions from the QTest library:

- QVERIFY - This allows the unit test to check that a certain condition was met and can be used in boolean comparisons.
- QCOMPARE - This function maps to AssertEquals in other testing libraries. It allows the comparison of two variables and passes the test if they are equal.

A generalized example of these functions can be seen in the documentation listed in **4.0 Testing in QT**.

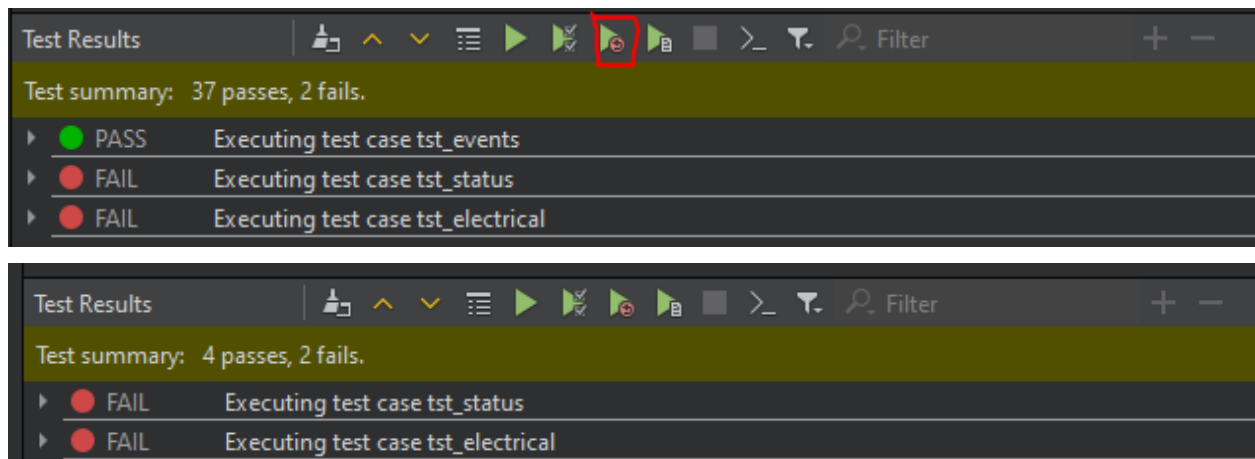
4.3 Running Tests

If the developer is using the QT Creator IDE, then the easiest option to run tests is located within QT Creator. Going to **Tools>Tests>select testing option**.

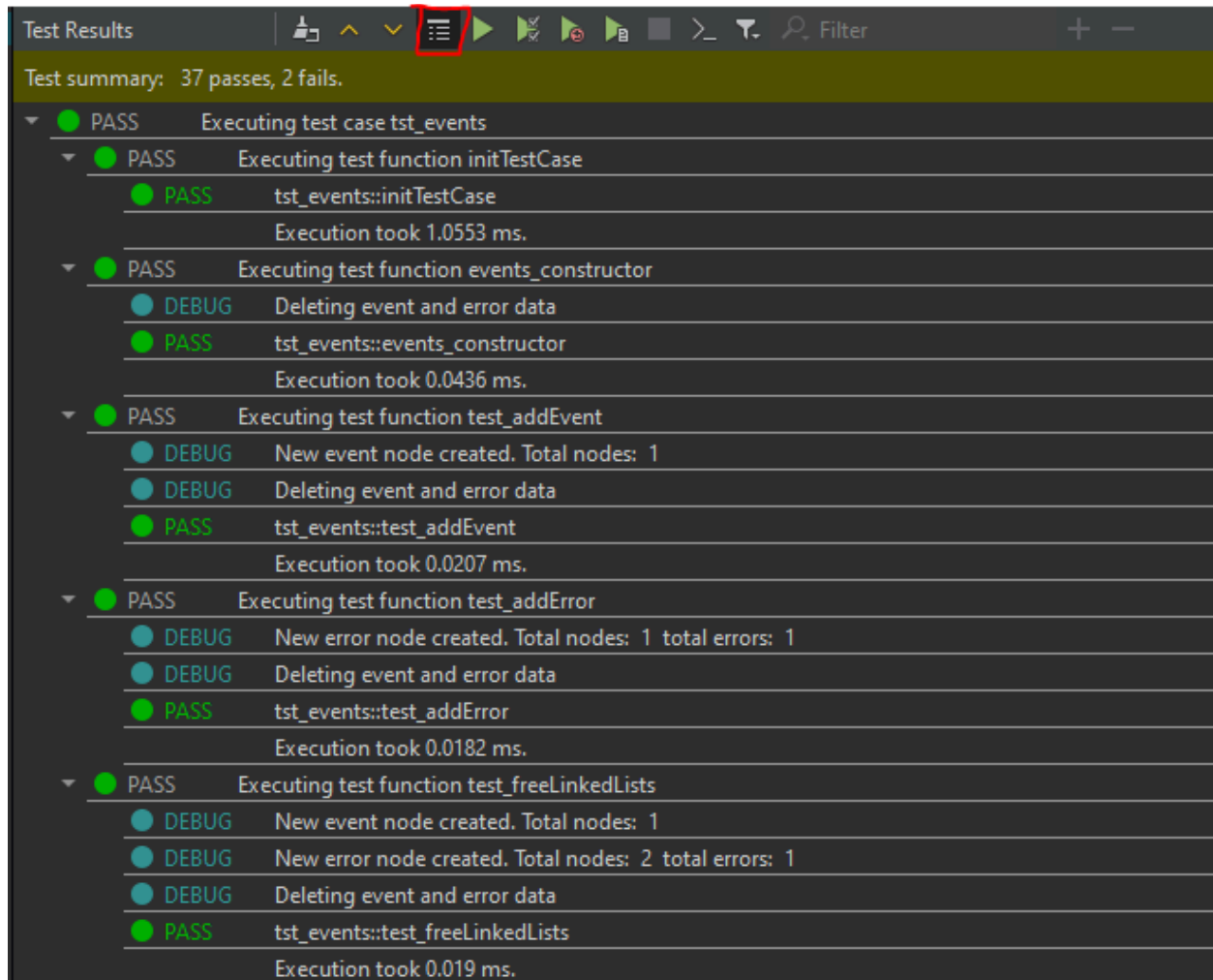


This option also allows keybinding the running of tests. Pressing **alt+shift+t**, then while holding **alt** pressing **A** will run the tests and display the test results.

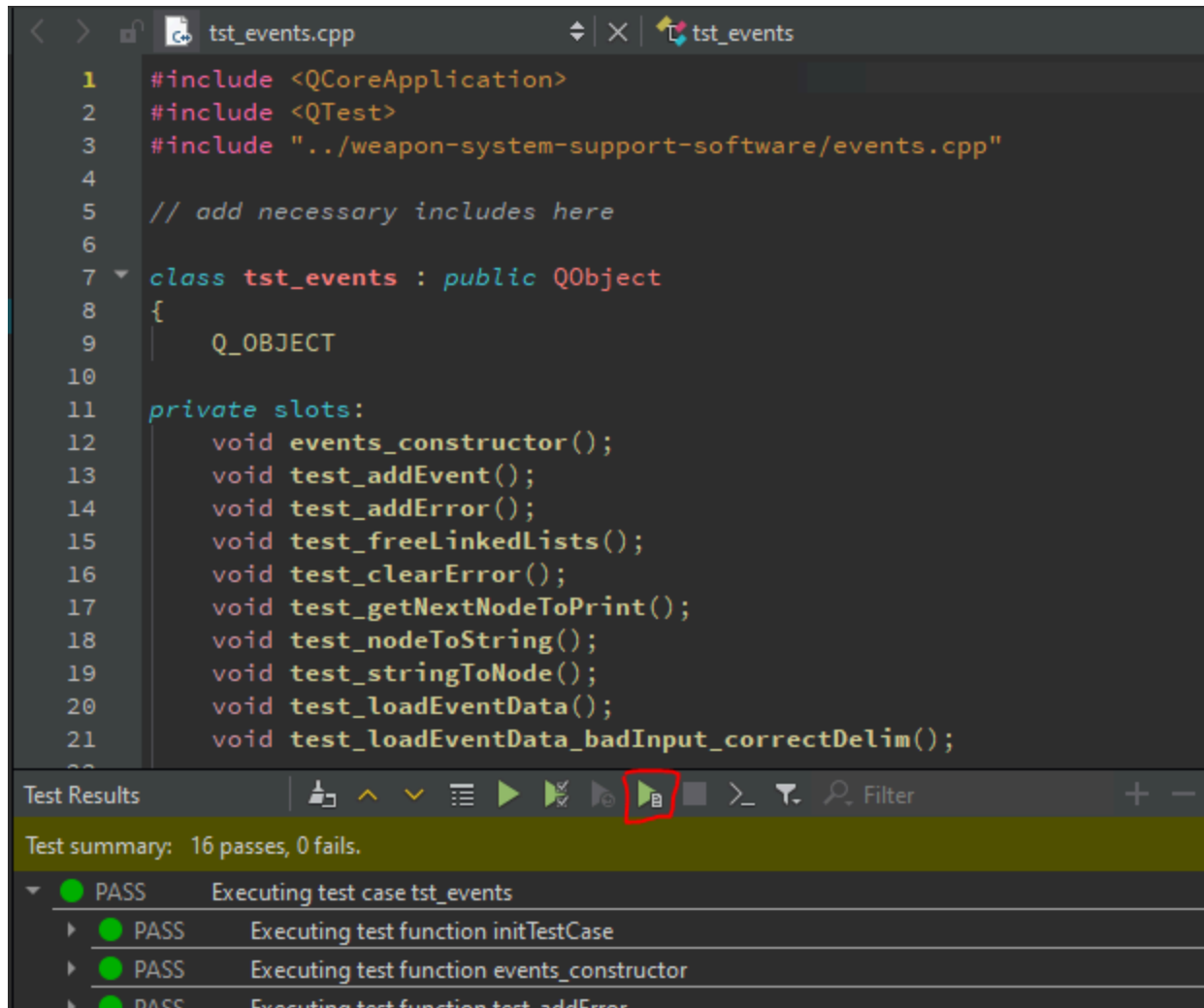
The developer can also specify which tests to run. In this instance, tests for the status and electrical classes are failing. By selecting the button below, the developer can run failed tests, the output of which will look like the second listed screenshot.



By selecting the highlighted button below, the developer will see the in depth information on the tests, including the individual checks and passes as well as any debug messages.



The developer can also target tests in specific files to be run. By opening the events tests file and selecting the highlighted button, the developer can target tests only for the events file.

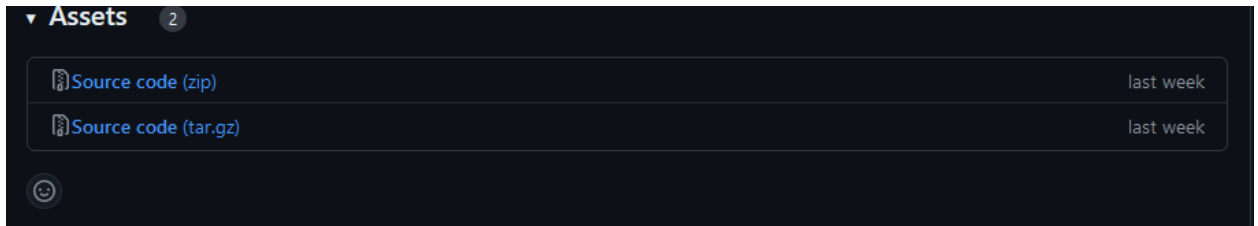


5.0 Packaging For Release

The team has packaged the software for release in two ways:

- After each sprint, an alpha release has been created, listing all changes and updates made to that version. These releases can be found here:
<https://github.com/Team-Controller-NAU/Controller/releases>.

To access these versions, simply install the source code as seen in this screenshot. It is important to note that many of these releases do not contain CMake files, so an existing CMake file will need to be modified to run the release.



- The team has been updating an installer for ease of use. The installer is made from the Inno Setup installer application. A link to the installer can be found here: <https://github.com/Team-Controller-NAU/Controller/blob/main/project-docs/WSSS%20Installer.exe>. For more information on the Inno Setup software, please refer to this link to the documentation: <https://jrsoftware.org/ishelp/>. Here is a video describing the basic steps to using the Inno Setup: <https://www.youtube.com/watch?v=wP9HcD4oPDw>